Proposing A New Task Model towards Many-Core Architecture

Akio Shimada[1], Balazs Gerofi[2], Atsushi Hori[1], and Yutaka Ishikawa[1,2]

[1] RIKEN Advanced Institute for Computational Science

a-shimada@riken.jp, ahori@riken.jp

[2] The University of Tokyo

bgerofi@il.is.s.u-tokyo.ac.jp, ishikawa@is.s.u-tokyo.ac.jp

ABSTRACT

Many-core processors are gathering attention in the areas of embedded systems due to their power-performance ratios. To utilize cores of a many-core processor in parallel, programmers build multi-task applications that use the task models provided by operating systems. However, the conventional task models cause some scalability problems when multi-task applications are executed on many-core processors. In this paper, a new task model named Partitioned Virtual Address Space (PVAS), which solves the problems, is proposed. PVAS enhances inter-task communications of multi-task applications and averts serialization of concurrent virtual memory operations. Preliminary evaluations by using micro benchmarks showed that PVAS has the potential to promote the performance of multi-task applications that run on many-core processors.

Keywords

Operating systems, Task model, Process, Thread

1. INTRODUCTION

Having reached the evolutionary limits of single-core performance in terms of power-efficiency, the number of cores being incorporated into multi-core processors has shown dramatic growth in recent years. By increasing the number of cores rather than promoting single-core performance, it is possible to achieve both high throughput and good power-performance ratios. Multi-core processors containing several tens or over hundred of cores are called *many-core* processors. Then, the utilization of many-core processors is becoming popular in the areas of embedded systems.

To efficiently utilize multiple cores in parallel, programmers commonly build multi-task applications using one of two task models, *multi-process* and *multi-thread*. However, those conventional task models are originally designed to implement time-shared multi-tasking for single-core processors, so some scalability problems arise when a large number of tasks are executed in parallel on a many-core processor. In this paper, how the conventional task models cause those problems is described. Then, a new task model that is capable of avoiding those problems is proposed. The proposed task model is named *Partitioned Virtual Address Space (PVAS)*, and preliminary evaluations conducted using micro benchmark tests show that our PVAS task model has the potential to promote the performance of multi-task applications that run on many-core processors.

MES'13 June 23 - 24 2013, Tel-Aviv, Israel

2. CURRENT TASK MODEL ISSUES

2.1 Multi-Process Model

In case of the multi-process model, a multi-task application invokes multiple processes and directs them to execute multiple tasks in parallel. During such executions, each task runs in an independent virtual address space. Multiple tasks within the same multi-task application must communicate with the others in order to exchange the data required for processing a parallel computation. Those inter-task communications must take place beyond address space boundaries between the tasks. To overcome the address space boundary, modern operating systems (OSes) provide a variety of inter-process communication (IPC) methods, such as socket, pipe, and shared memory. In general, those IPC methods are implemented by using intermediate buffers located on the kernel space or in shared memory regions. In those methods, two memory copies (from sender task to inter-mediate buffer and from intermediate buffer to receiver task) take place for every inter-task communication. These two memory copies result in high latency which can then result in performance bottlenecks for multi-task applications. This problem can be more serious for multi-task applications that run on many-core processors because a larger number of processes can be invoked, and the frequency at which the inter-task communication takes place can be higher.

In the case of N tasks, $N(N-1)/2$ communication patterns exist. So, to execute the communications between tasks in the most efficient way, the multi-task application has to preliminarily create $N(N-1)/2$ intermediate buffers. By doing so, all possible communication patterns can be executed simultaneously, and the multi-task application can

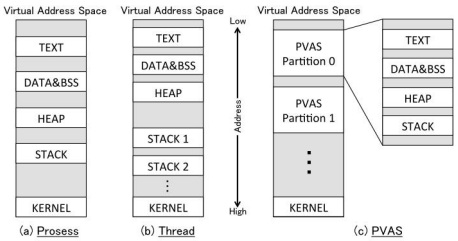

Figure 1: Semantic views of the virtual address space of the conventional task models and PVAS task model

achieve efficient parallelization. However, this results in much memory consumption. For example, in the case of 100 tasks, 4950 buffers are required. Even if the size of each buffer is only four kilobytes, approximately two gigabytes of physical memory in total are consumed for the inter-task communication. This memory consumption can be a serious problem for embedded systems that have only a few gigabyte of main memory at the most.

2.2 Multi-Thread Model

In case of the multi-thread model, multiple threads are invoked within a process, which then execute tasks in parallel. On this model, all tasks related to a multi-task application run in the same virtual address space. They can access the data of the other task directly whenever necessary because there are no address space boundaries between the tasks. As a result, high-performance inter-task communication can be achieved without inter-mediate buffers. However, another problem, the serialization of the virtual memory (VM) operations, arises on this model.

Modern OS kernels manage a virtual address space using a *memory region tree* and a *page table tree*. Those trees are shared by all tasks running in the same virtual address space, and VM operations (`mmap`, `munmap` and page fault handling) issued by the multiple tasks update those trees. The memory region tree stores information regarding memory mappings within a virtual address space. The `mmap` operation creates a new memory mapping and stores its information in the memory region tree. The `munmap` operation deletes any memory mapping and removes its related information from the memory region tree. If multiple tasks running in the same virtual address space execute those VM operations simultaneously, they must be serialized in the kernel to avoid the race condition that would occur during the memory region tree update process.

The page table tree manages mappings from physical to virtual pages. If any task incurs a page fault, a page fault handler looks up the faulting address in the memory region tree. If the faulting address is included in any memory mapping, the page fault handler then allocates a new page and maps it to the virtual address space by recording the new page mapping on the page table tree. If multiple tasks running in the same virtual address space incur page fault handling operations simultaneously, those operations must be serialized in the kernel to avoid the race condition that would occur during the page table tree update.

These serializations described above block parallel processing and limit the performance of multi-task applications [5]. This problem can be more serious for multi-task applications that run on many-core processors because a larger number of threads can be created, and the frequency at which the race condition occurs can be higher.

3. PVAS TASK MODEL

3.1 Address Space Design

Figure 1(a) shows the design of the virtual address space on the multi-process model. On this model, each task runs in an independent virtual address space, and each task has its own TEXT/DATA&BSS/HEAP segments within its virtual address space. Each task uses its DATA&BSS/HEAP segments as its private data regions. There are address space boundaries between the tasks constructing a multi-task application.

Figure 1(b) shows the design of the virtual address space on the multi-thread model. On this model, multiple tasks run in the same virtual address space and STACK segments are prepared for each task. The TEXT segment is shared by all tasks running in the same virtual address space. The DATA&BSS/HEAP segments are also shared by all tasks.

Figure 1(c) shows our design of the virtual address space on the PVAS task model proposed in this paper. The idea behind the PVAS task model is to partition a virtual address space and assign one partition to one task. On the PVAS task model, multiple tasks run in the same virtual address space. A task on the PVAS task model is called *PVAS task*. The virtual address space for this model is partitioned, and each partitioned region is called a *PVAS partition*. Each PVAS task is assigned its own PVAS partition, which also contains its TEXT/DATA&BSS/HEAP/STACK segments, and PVAS tasks use their DATA&BSS/HEAP segments as their private data regions. While each PVAS task accesses the data located on its DATA&BSS/HEAP segments, it can also access the data of the other PVAS task directly whenever necessary because there are no address space boundaries between the PVAS tasks. As a result, high-performance inter-task communication can be achieved without inter-mediate buffers.

3.2 Virtual Memory Management

Because multiple tasks run in the same virtual address space on the PVAS task model, it was initially thought that VM operations simultaneously issued by multiple tasks would need to be serialized in the kernel as well as the multi-thread model. To avoid this problem, multiple memory region trees are created to manage PVAS partitions instead of creating a single memory region tree to manage the whole virtual address space, as described in Figure 2. On the PVAS task model, each PVAS task can create memory mappings within its PVAS partition and cannot create memory mappings within the PVAS partition dedicated to any other PVAS task. Thus, each PVAS task updates only one memory region tree which manages its PVAS partition. As a result, VM operations such as `mmap` and `munmap` do not need to be serialized when the memory region tree is updated, even if multiple PVAS tasks execute those VM operations simultaneously.

The procedure of the page fault handling operation on the PVAS task model is as follows. If any PVAS task incurs a page fault, a page fault handler confirms which PVAS partition the faulting address belongs to, and looks up the faulting address in the memory region tree managing the PVAS partition to which the faulting address belongs. If

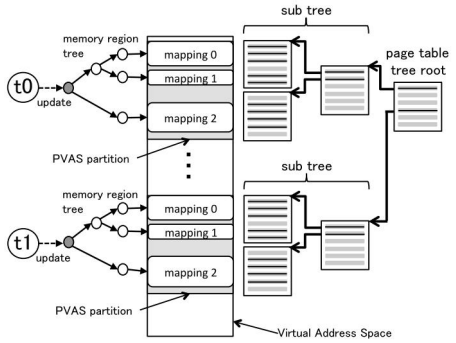

Figure 2: Semantic view of the VM management on the PVAS task model. Each parallel task (t0, t1) updates only its own memory region tree.

the faulting address is included in any memory mapping, the page fault handler then allocates a new page and maps it to the virtual address space by recording the new page mapping on the page table tree. All PVAS tasks running in the same virtual address space share the same page table tree. However it is logically partitioned into sub trees. Each sub tree covers the address range of one of the PVAS partitions as describe in Figure 2. If multiple PVAS tasks try to update the same sub tree simultaneously, these operations are serialized to avoid the race condition. In this scheme, the serialization of page fault handling operations may occur when any PVAS task accesses the data located on the PVAS partition dedicated to any other PVAS task. However, each PVAS task accesses the data within its PVAS partition basically. Thus, the frequency at which the serialization of page fault handling operations occurs can be lower on the PVAS task model.

4. PRELIMINARY EVALUATION

Currently, PVAS is implemented in Linux kernel version 2.6.32 for the x86-64 architecture. Our implementation of PVAS was evaluated by a ping-pong communication benchmark implemented by the message passing interface (MPI), and a handmade VM operation benchmark. The evaluations were performed with Intel Xeon X5670 multi-core processor (6 cores × 2 Sockets).

4.1 Inter-task Communication

MPI is a standard specification that is widely used in HPC applications. MPI provides the interface for the message passing communication to the multi-task application built from the multi-process model. MPICH2 [1] is one of the MPI implementations. MPICH2 uses Nemesis [4], which is a low-level inter-task communication layer. Here, MPICH2 was modified to support the PVAS task model. The task manager of modified MPICH2 invokes the PVAS tasks, which execute the MPI program. Furthermore, Nemesis was also modified to leverage the inter-task communication of PVAS. The original Nemesis performs inter-task communication by using the shared memory as the inter-mediate buffer, which produces two memory copies for every communication. On the PVAS implementation, the sender task directly copies data from its send buffer to the receive buffer of the receiver task, so only one memory copy is produced for every communication. If the modified Nemesis shows better performance than does the original version, we can say that PVAS has the potential to improve inter-task communication of the multi-task application.

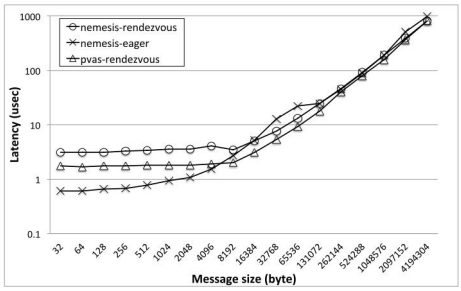

Figure 3: The performance of the ping-pong latency

The ping-pong benchmark of the Intel MPI Benchmarks [6] was used for this evaluation. The PVAS implementation that supports the rendezvous protocol and the original Nemesis implementation that supports both the eager and the rendezvous protocols were evaluated. MPI message communication uses two protocols: the rendezvous protocol and the eager protocol. In the rendezvous protocol, a sender task and a receiver task are synchronized until both tasks are ready. In the eager protocol, the sender task sends messages regardless of the state of the receiver task. With this protocol, the sender task has a buffer for send operations. After copying the send data to the buffer, the sender task can start to execute the next operation.

Figure 3 shows the latency of the ping-pong communication of a pair of tasks. When the message size is sufficiently smaller than the buffer size (smaller than eight kilobyte), the Nemesis eager protocol shows the best performance. The PVAS implementation shows better performance than the Nemesis rendezvous protocol, because the PVAS implementation produces only one memory copy for every inter-task communication, while the Nemesis rendezvous protocol produces two memory copies. When the message size is bigger than 256 kilobytes, all implementations show almost the same performance, because the Nemesis implementation splits a message when the message size is big, and sends the split messages in the same manner as in pipeline processing.

Also the PVAS implementation can support the eager protocol. If the PVAS implementation adopts the eager protocol when the message size is small, it can achieve higher performance than the original Nemesis in all cases.

4.2 VM Operation

To evaluate the VM operation performance of the PVAS task model, a micro benchmark was performed. This benchmark invokes multiple tasks that execute VM operations in parallel. In this benchmark, each task executes `mmap`, `memset` and `munmap` 1000 times. 1 MB of the memory region is allocated by `mmap`, and accessed by `memset`, so page fault handling operations are executed. The allocated memory region is deleted by `munmap`. The benchmark is implemented by the conventional task models and the PVAS task model.

The result of this benchmark is shown in Figure 4. In this graph, the horizontal axis represents the number of parallel tasks, and the vertical axis represents the average number of CPU cycles counted by the time stamp counter.

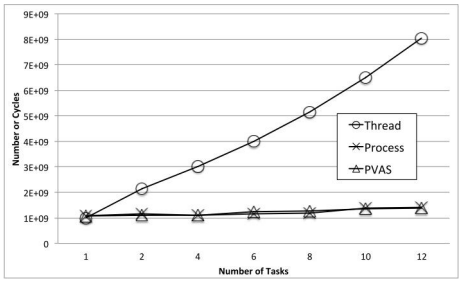

Figure 4: CPU cycles in 1000 time mmap, memset, and munmap calls

In the graph, the performance of the PVAS task model is comparable with the multi-process model, while the multi-thread model shows poor performance. On the multi-process model, each task has its own virtual address space, so concurrent VM operations are not serialized. As described in Section 3.2, the concurrent `mmap` and `munmap` operations are not serialized on the PVAS task model as well as on the muti-process model. In addition, page fault handling operations are not serialized in this benchmark because the inter-task communication does not take place. This is because both of them show good performance. In contrast, on the multi-thread model, VM operations need to be serialized as described in Section 2.2. This serialization incurred the bad performance.

5. RELATED WORK

5.1 Kernel Thread Copy

Another method called *inter-task communication via the kernel thread* [7] was proposed to improve the inter-task communication of the multi-task application built from the multi-process model. Since the kernel thread can access the data of all user processes beyond the address space boundary, inter-task communication via the kernel thread produces only one memory copy. Therefore, the communication latency is smaller than that occurring during inter-task communication using the intermediate buffer (where two memory copies are created). However, this method via the kernel thread introduces overhead in the form of context switching between the user processes and the kernel thread. As a result, the communication latency becomes larger than that of PVAS.

5.2 Virtual Address Space Mapping

XPMEM [2] is a Linux kernel module that enables any process to map memory pages of another process into its virtual address space. After mapping, the process can then access them in the same way as it accesses the local data. This XPMEM capability improves the communication performance of a multi-task application built from the multi-process model. However, XPMEM needs an extra memory mapping operation. This mapping operation causes the overhead.

SMARTMAP [3] enables a process for mapping the memory of another process into its virtual address space, in a way similar to that of XPMEM. In SMARTMAP, a process occupies an address range from 0 to 512 GB as its local address space. The rest of the virtual address space is treated as a global address space and is used for mapping the memory of all other processes. Thus, a process can directly access the memory of another process. Each process of SMARTMAP has a separate page table tree, but the portion of the page table tree that maps the global address space is shared by all processes in SMARTMAP, so page fault handling operations need to be serialized as on the multi-thread model. To avoid serialization of the page fault handling, SMARTMAP does not support on-demand paging and uses linear mapping from the virtual addresses to the physical pages of the memory. This incurs poor memory-use efficiency.

6. CONCLUSION AND FUTURE WORK

This paper proposed a new task model, called PVAS, that is designed to promote the performance of the multi-task applications that run on many-core processors. PVAS enhances the inter-task communication of a multi-task application. On the PVAS task model, inter-task communication can take place with only one memory copy, and it does not require inter-mediate buffers that incur huge memory consumption. Moreover, the virtual memory management of PVAS averts the serialization of concurrent VM operations.

Preliminary evaluations showed that PVAS has the potential to increase the performance of multi-task applications that run on many-core processors. The evaluation of the ping-pong communication benchmark showed that PVAS has the capability to improve the latency of the inter-task communication. The evaluation of the VM operation benchmark showed that PVAS has the capability of achieving efficient parallelization, even if the number of tasks increases. Preliminary evaluations were performed in the small environment by using micro benchmarks. The ping-pong communication benchmark was performed with only two tasks, and the VM benchmark was performed with only twelve tasks. Evaluation of PVAS by using real applications in the large environment is future work.

7. ACKNOWLEDGMENTS

This research was partially supported in part by the CREST project of Japan Science and Technology Agency (JST).

8. REFERENCES

[1] MPICH2. `http://www.mcs.anl.gov/research/projects/mpich2/`.

[2] . XPMEM Cross-Process Memory Mapping. `http://code.google.com/p/xpmem/`.

[3] R. Brightwell, K. Pedretti, and T. Hudson. SMARTMAP: operating system support for efficient data sharing among processes on a multi-core processor. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 25:1–25:12, Piscataway, NJ, USA, 2008. IEEE Press.

[4] D. Buntinas, G. Mercier, and W. Gropp. Design and Evaluation of Nemesis, a Scalable, Low-Latency, Message-Passing Communication Subsystem. In *Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid*, CCGRID '06, pages 521–530, Washington, DC, USA, 2006. IEEE Computer Society.

[5] A. T. Clements, M. F. Kaashoek, and N. Zeldovich. Scalable address spaces using rcu balanced trees. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 199–210, New York, NY, USA, 2012. ACM.

[6] Intel Corporation. Intel MPI Benchmarks 3.2.3. `http://software.intel.com/en-us/articles/intel-mpi-benchmarks/`.

[7] T. Takahashi, F. O'Carroll, H. Tezuka, A. Hori, S. Sumimoto, H. Harada, Y. Ishikawa, and P. H. Beckman. Implementation and Evaluation of MPI on an SMP Cluster. In *IPPS/SPDP Workshops*, pages 1178–1192, 1999.